

P01A: Entorno de pruebas

IMPORTANTE. A TENER EN CUENTA DURANTE TODO EL CURSO

- Debes subir a Bitbucket las prácticas que realices. **SÓLO SE REVISARÁN** aquellas prácticas subidas **antes** de iniciar la sesión de la siguiente práctica.
- **DURANTE** las clases de **prácticas**:
 - con **carácter general** se darán explicaciones sobre las soluciones de la práctica anterior,
 - con **carácter individual** se realizará el seguimiento del trabajo subido por el alumno (siempre y cuando se haya hecho dentro del plazo establecido) y
 - se resolverán las dudas que surjan.
- Cada alumno tiene asignado un profesor de prácticas, que realizará el seguimiento de vuestro trabajo. No se admitirán cambios de turnos de prácticas que no sean realizados a través de la secretaría del centro.
- Recuerda que tu trabajo de prácticas te permitirá comprender y asimilar los conceptos vistos en las clases de teoría. Y que vamos a evaluar no sólo tus conocimientos teóricos sino que sepas aplicarlos correctamente. Por lo tanto, el **resultado** de tu trabajo **PERSONAL** sobre las clases en aula y en laboratorio determinará si alcanzas o no las competencias teórico-prácticas planteadas a lo largo del cuatrimestre.

Una vez que tengamos la máquina virtual preparada, nuestro repositorio Git creado y clonado en la máquina virtual, y nuestra licencia JetBrains activa, ya estamos en disposición de comenzar con nuestro trabajo práctico.

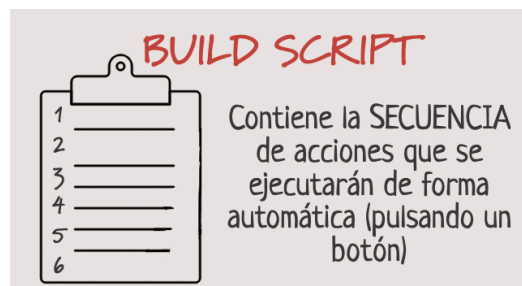
Maven

Maven es una **herramienta de construcción** de proyectos Java. Por definición, la construcción (*build*) de un proyecto es la secuencia de tareas que debemos realizar para, a partir del código fuente, poder usar (ejecutar) nuestra aplicación. Ejemplos de tareas que forman parte del proceso de construcción pueden ser compilación, *linkado*, pruebas, empaquetado, despliegue.... Otros ejemplos de herramientas de construcción de proyectos son *Make* (para lenguaje C), *Ant* y *Graddle* (también para lenguaje Java).

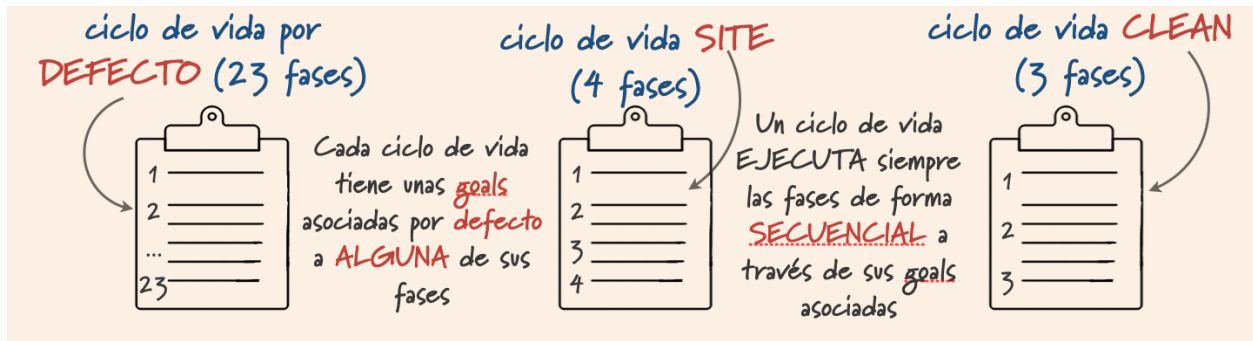
Maven puede utilizarse tanto desde línea de comandos (comando mvn) como desde un IDE.

Cualquier herramienta de construcción de proyectos necesita conocer la secuencia de tareas que debe ejecutar para construir un proyecto. Dicha secuencia de tareas se denomina **Build Script**.

Maven, a diferencia de Make o Ant (Graddle utiliza elementos de Ant y de Maven), permite definir el *build script* para un proyecto de forma declarativa, es decir, que no tenemos que indicar de forma explícita la “lista de tareas” a realizar, ni “invocar” de forma explícita la ejecución de dichas tareas.



Maven tiene predefinidas TRES secuencias de tareas (*build scripts*) para construir proyectos. Cada una de estas secuencias se denomina **ciclo de vida**.



El ciclo de vida más utilizado es el ciclo de vida por defecto (**default lifecycle**), y está formado por una lista de 23 tareas, denominadas **fases**. Ejemplos de fases son: *compile*, *test*, *package*, *deploy*,... Es importante que tengas claro que una fase es un concepto LÓGICO, no es un ejecutable, sino que podrá tener ASOCIADO algún ejecutable que realice una determinada acción.

FASES maven
(cada fase puede tener asociadas cero o más acciones ejecutables)

Una fase Maven identifica cuál debe ser la naturaleza de la acción o acciones que se ejecuten DURANTE la misma. Por ejemplo, el ciclo de vida por defecto contiene la fase "compile" y la fase "test": la primera permite asignar acciones ejecutables que lleven a cabo el proceso de compilación del proyecto, mientras que la segunda está pensada para que se ejecuten las pruebas unitarias (lógicamente, la fase de compilación será anterior a la fase de pruebas).

GOALS y plugins
(una goal puede asociarse a una fase.
Un plugin tiene 1 o varias goals)

Las acciones que se ejecutan en cada una de las fases se denominan **GOALS**. Por ejemplo la fase compile tiene asociada por defecto la **goal (o acción)** denominada *compiler:compile*, que lleva a cabo la compilación de los fuentes del proyecto. Cualquier goal pertenece a un **PLUGIN**. Un plugin no es más que un conjunto de goals. Por ejemplo, el plugin *compiler* contiene las goals *compiler:compile* y *compiler:testCompile* (el nombre de la goal SIEMPRE va precedida del nombre del *plugin* separado por ":",)

El proceso de construcción de maven genera un fichero empaquetado (jar, war, ear, ...), en el directorio *target*. Cada tipo de empaquetado, tiene asociadas **POR DEFECTO ciertas GOALS**.

Por ejemplo, cuando nuestro proyecto se empaqueta como un .jar, las GOALS asociadas al ciclo de vida por defecto son las siguientes:

Fase	plugin : goal	acciones
process-resources	maven-resources-plugin: resources	Copia *.* de /src/main/resources en target
compile	maven-compiler-plugin: compile	Compila *.java de /src/main/java
process-test-resources	maven-resources-plugin: testResources	Copia *.* de /src/test/resources en target
test-compile	maven-compiler-plugin: testCompile	Compila *.java de /src/test/java
test	maven-surefire-plugin: test	Ejecuta los tests unitarios
package	maven-jar-plugin: jar	Empaqueta *.class + recursos en un jar
install	maven-install-plugin: install	Copia el fichero jar en repositorio local
deploy	maven-deploy-plugin: deploy	Copia el fichero jar en repositorio remoto

Una **goal**, por tanto, no es más que un código ejecutable, implementado por algún desarrollador. Algunos desarrolladores "deciden" que una determinada goal estará asociada POR DEFECTO a una determinada fase de algún ciclo de vida Maven. Todas las goals son **CONFIGURABLES** (disponen de un conjunto de variables propias que tienen valores por defecto y que podemos cambiar). Por ejemplo, podemos cambiar la fase a la que se asociará dicha goal. Una goal SIEMPRE está contenida en un PLUGIN.

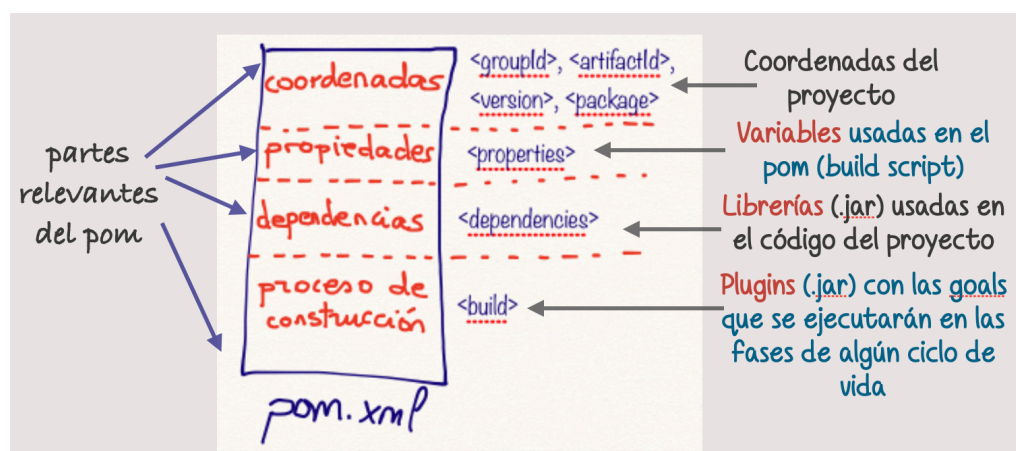
La forma de provocar la ejecución de una goal durante una fase consiste simplemente en añadir el plugin que la contiene en el fichero pom.xml (y configurar sus valores por defecto, si es necesario). Si una goal no tiene asociada una fase por defecto, y no asociamos de forma explícita dicha goal a alguna fase, la goal **NO SE EJECUTARÁ** (aunque incluyamos su plugin en el fichero pom.xml).

Por ejemplo, cuando el empaquetado es **jar**:

- ❖ La GOAL *compiler:testCompile* se ejecutará automáticamente durante la fase test-compile.
- ❖ La GOAL *compiler:compile* se ejecutará automáticamente durante la fase compile
- ❖ NO es necesario incluir el plugin **compiler** en el pom, a menos que queramos cambiar su configuración por defecto.

Cualquier proyecto Maven debe contener en su directorio raíz el fichero **pom.xml**. Dicho fichero nos permitirá configurar la secuencia de acciones a realizar (build script) para construir el proyecto, mediante la etiqueta <build>. También podremos indicar qué librerías (ficheros .jar) son necesarias para compilar/ejecutar/probar... nuestro proyecto (etiqueta <dependencies>).

Es importante que sepamos identificar al menos 4 "secciones" en el fichero pom.xml. Cada una de ellas se caracteriza por usar determinadas etiquetas:



Como resultado del proceso de construcción de un proyecto Maven se obtiene un **artefacto** (fichero empaquetado) que se identifica mediante sus **coordenadas**, separadas por ":". Para identificar un artefacto Maven se utilizan, como mínimo, tres elementos, o coordenadas, de forma que cualquier artefacto Maven se especifica de forma única (no hay dos artefactos con las mismas coordenadas).

Las coordenadas obligatorias que identifican de forma única a un artefacto Maven son: **groupId:artifactId:version**, en donde:

- ❖ **groupId** es el identificador de grupo. Se utiliza normalmente para identificar la organización o empresa desarrolladora y puede utilizar notación de puntos. Por ejemplo: *org.ppss*
- ❖ **ArtifactId** es el identificador del artefacto (nombre del archivo), normalmente es el mismo que el nombre del proyecto. Por ejemplo: *practica1*

- ❖ **Version** es la versión del artefacto. Indica la versión actual del fichero correspondiente. Por ejemplo: *1.0-SNAPSHOT*.

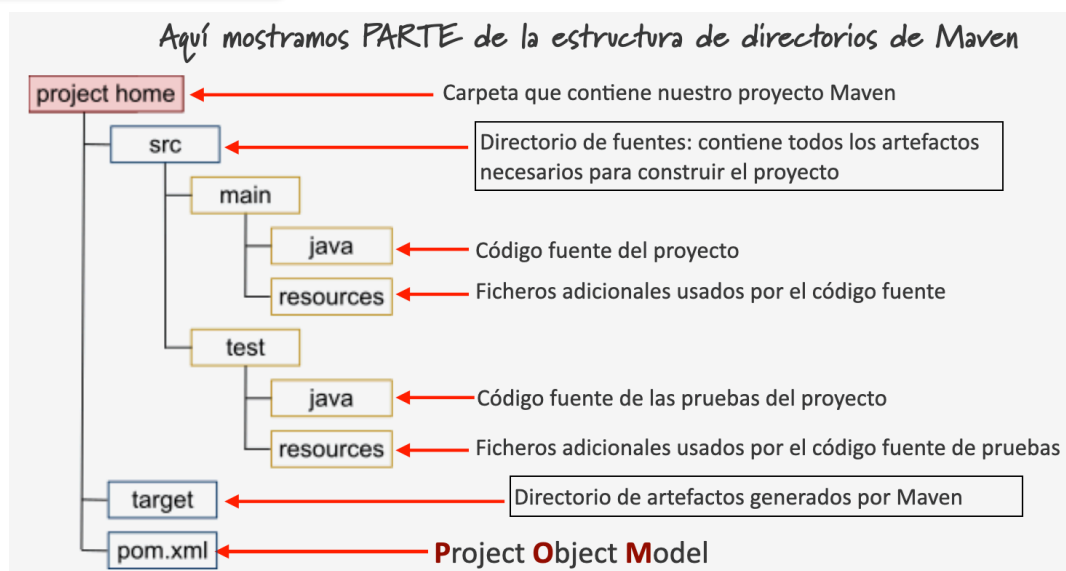
Los artefactos se almacenan en un repositorio local Maven, situado en `$HOME/.m2/repository`. Las coordenadas se usan para identificar exactamente la ruta de cada fichero en el repositorio maven. Por ejemplo:

- ❖ ***org.ppss:practica1:1.0-SNAPSHOT*** representa al fichero `$HOME/.m2/repository/org/ppss/practica1/1.0-SNAPSHOT/practica1-1.0-SNAPSHOT.jar` (por defecto, los artefactos tienen la extensión `.jar`)
- ❖ ***org.ppss:practica1:2.0-SNAPSHOT*** representa al fichero `$HOME/.m2/repository/org/ppss/practica1/2.0-SNAPSHOT/practica1-2.0-SNAPSHOT.jar`
- ❖ ***org.ppss:proyecto3:war:1.0-SNAPSHOT*** representa al fichero `$HOME/.m2/repository/org/ppss/proyecto3/1.0-SNAPSHOT/proyecto3-1.0-SNAPSHOT.war`

estructura de directorios Maven

(la misma en *TODOS* los proyectos Maven)

Todos los proyectos Maven usan la MISMA estructura de directorios. Así, por ejemplo, el código fuente del proyecto estará en el directorio **src/main/java**, y el código que implementa las pruebas del proyecto siempre lo encontraremos en el directorio **src/test/java**. Los artefactos generados durante la construcción, por ejemplo los ficheros `.class`, siempre estarán en el directorio **target** (o alguno de sus subdirectorios). El directorio **target** se genera automáticamente en cada construcción del proyecto, por eso no necesitamos "guardarlo" en Bitbucket.



Por otro lado, cualquier LIBRERÍA EXTERNA (ficheros `.jar`) que utilicemos en nuestro proyecto Maven, debe incluirse en el fichero `pom.xml` (en la sección **<dependencies>**), de forma que, si no se encuentra físicamente dicho fichero en nuestro disco duro, Maven lo descarga automáticamente de sus repositorios en la nube. Es más, si utilizamos una librería, que a su vez depende de otra, Maven automáticamente se encarga de descargarse también esta última, y así sucesivamente. Esto hace que nuestros proyectos "pesen" poco, ya que tanto el directorio **target**, como cualquier librería y/o plugin utilizados por el proyecto se descargarán y/o generarán automáticamente si es necesario, cada vez que construyamos el proyecto. Por tanto, si queremos "llevarnos" nuestro proyecto a otra máquina, únicamente necesitamos el fichero `pom.xml`, y el directorio **src** del proyecto (el directorio **src** contiene todos los fuentes del proyecto).

**repositorios
locales y
remotos Maven**
*(almacenan
artefactos
Maven)*

Todos los artefactos generados y/o utilizados por Maven se almacenan en repositorios. Maven mantiene una serie de repositorios remotos, que alojan todos los plugins y librerías que podemos utilizar. Cuando ejecutamos Maven por primera vez en nuestra máquina, se crea el directorio **.m2** (en nuestro \$HOME), que será nuestro repositorio local. Cuando iniciamos un proceso de construcción Maven, primero se consulta nuestro repositorio local, para ver si contiene todos los artefactos necesarios para realizar la construcción. Si falta algún artefacto en nuestro repositorio local, Maven automáticamente lo descargará de algún repositorio remoto.

**Ejecución de
Maven**
(mvn fase/goal)

Para iniciar el proceso de construcción de Maven, usamos el comando **mvn** seguido de la fase (o fases) que queramos realizar, o bien indicando la goal, o goals que queremos ejecutar de forma explícita (separadas por espacios). Si incluimos una o más fases, o goals, se ejecutarán una por una en el mismo orden que hemos indicado. Por ejemplo, si tecleamos: **mvn fase1 fase2 plugin1:goal3 plugin2:goal4**, será equivalente a ejecutar: **mvn fase1, mvn fase2, mvn plugin1:goal3, y mvn plugin2:goal4**, en este orden.

El comando **mvn <faseX>** ejecuta todas las goals asociadas a todas y cada una de las fases, siguiendo exactamente el orden de las mismas en el ciclo de vida correspondiente, desde la primera, hasta la fase que hemos indicado (<faseX>).

El comando **mvn plugin:goal** ejecuta únicamente la goal que hemos especificado

Las fases se ejecutan siempre en el mismo orden
comenzando desde la PRIMERA !!!

FASES	PLUGIN : GOALS
1 validate	
2 initialize	
3 generate-sources	
4 process-sources	
5 generate-resources	
6 process-resources	resources:resour
7 compile	compiler:compile
8 process-classes	
9 generate-test-sources	
10 process-test-sources	
11 generate-test-resources	
12 process-test-resources	resources:testResources
13 test-compile	compiler:testCompile
14 process-test-classes	
15 test	surefire:test
16 prepare-package	
17 package	jar:jar
18 pre-integration-test	
19 integration-test	
20 post-integration-test	
21 verify	
22 install	install:install
23 deploy	deploy:deploy

mvn test-compile

Ejecuta las goals de las fases 1..13
Genera los .class de src/test/java

mvn test

Ejecuta las goals de las fases 1..15
Ejecuta los .class de src/test/classes y genera un informe

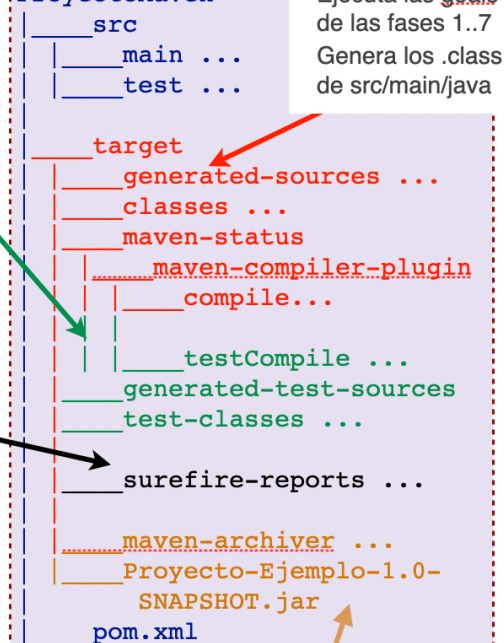
mvn package

Ejecuta las goals de las fases 1..17
Genera el .jar del proyecto



EJEMPLO

ProyectoMaven



mvn compile

Ejecuta las goals de las fases 1..7
Genera los .class de src/main/java

IntelliJ IDEA Ultimate

IntelliJ es un IDE muy utilizado para trabajar con diferentes tipos de aplicaciones, entre ellas aplicaciones java y Maven. Trabajaremos SIEMPRE con proyectos Maven.

En esta primera práctica empezaremos a familiarizarnos con el uso de esta herramienta. Veamos primero algunos conceptos importantes:

- **Project.** Todo lo que hacemos con IntelliJ IDEA se realiza en el contexto de un **Proyecto**. Los proyectos no contienen en sí mismos artefactos tales como código fuente, *scripts* de compilación o documentación. Son el nivel más alto de organización en el IDE, y contienen la definición de determinadas propiedades. Para los que estéis familiarizados con Eclipse, un proyecto sería similar a un *workspace* de Eclipse. La configuración de los datos contenidos en un proyecto se puede almacenar en un directorio denominado **.idea**, y es creado y mantenido automáticamente por IntelliJ.
- **Module.** Un Módulo es una unidad funcional que podemos compilar, probar y depurar de forma independiente. Los módulos contienen, por lo tanto, artefactos tales como código fuente, scripts de compilación, tests, descriptores de despliegue, y documentación. Sin embargo un módulo no puede existir fuera del contexto de un proyecto. La información de configuración de un módulo se almacena en un fichero denominado **.iml**. Por defecto, este fichero se crea automáticamente en la raíz del directorio que contiene dicho módulo. Un proyecto IntelliJ puede contener uno o varios módulos. Para los que estéis familiarizados con Eclipse, un módulo sería similar a un *proyecto* de Eclipse
- **Facet.** Las **Facetas** representan varios *frameworks*, tecnologías y lenguajes utilizados en un módulo. El uso de facetas permite descargar y configurar los componentes necesarios de los diferentes frameworks. Un módulo puede tener asociadas varias facetas. Algunos ejemplos de facetas son: Android, AspectJ, EJB, JPA, Hibernate, Spring, Struts, Web, Web Services,...
- **Run/Debug Configuration.** Podemos configurar la ejecución de determinadas acciones (como por ejemplo arrancar/parar un servidor de aplicaciones, lanzar un *script* de compilación, ...), de forma que quede guardada con un determinado nombre y la podamos lanzar a voluntad, simplemente con un *click* de ratón. IntelliJ tiene varias configuraciones predefinidas, y podemos crear nuevas configuraciones a partir de éstas. Para los que estéis familiarizados con Eclipse, una configuración de ejecución en IntelliJ sería similar al mismo concepto en Eclipse.

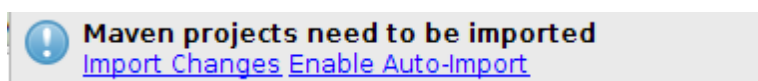
⇒ Creación de un proyecto IntelliJ a partir de un proyecto Maven existente

Una posible forma de hacerlo es a partir del **fichero pom.xml** presente en cualquier proyecto Maven. Para ello simplemente:

- Desde el menú principal elegimos File→Open (u opción Open cuando abrimos IntelliJ)
- En el cuadro de diálogo seleccionamos el fichero pom.xml, y pulsamos OK. Nos preguntará si queremos abrir como fichero o como proyecto. Elegiremos como proyecto.

De forma alternativa, también podremos usar la opción File→Open y seleccionar la **carpeta** que contiene el fichero pom.xml

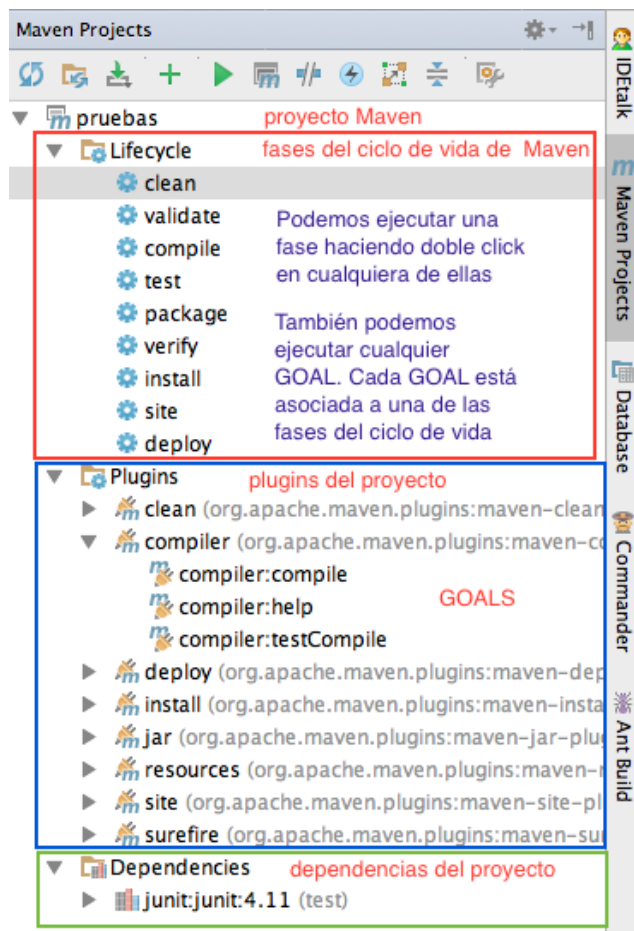
Importante: siempre que editemos la configuración que afecte de alguna manera a la construcción de nuestro proyecto (por ejemplo editando directamente el pom.xml, cambiando la ubicación y/o versión de maven del sistema...) nos aparecerá el siguiente cuadro de diálogo:



Si marcamos “enable Auto-import”, los cambios se importarán siempre de forma automática.

⇒ IntelliJ IDEA Maven Tool Window

IntelliJ permite mostrar diferentes **“Tool Windows”** que permiten mostrar diferentes perspectivas del proyecto. Una de estas “vistas” es la ventana de Maven (**“Maven tool window”**), que podremos mostrar si estamos trabajando con un proyecto Maven. Para mostrar la ventana tenemos que hacerlo desde **View→Tools Windows→Maven**. A continuación mostramos el aspecto de dicha ventana.



Dentro del elemento **Lifecycle**, vemos un subconjunto de fases (de los tres ciclos de vida que proporciona Maven).

Hacer doble click sobre cualquiera fase equivale a ejecutar el comando **mvn <faseX>**. Como resultado se ejecutarán todas las *goals* desde la primera fase hasta <faseX> que estén asociadas a cada una de ellas.

El elemento **Plugins** nos muestra los plugins asociados a las fases mostradas en el elemento Lifecycle por nuestro proyecto. Si añadimos algún plugin en nuestro pom también se mostrará, así como las goals que contiene. Podemos ejecutar el comando **mvn <goal>** haciendo doble click sobre cualquiera de ellas.

Podemos observar también (en gris) las coordenadas de cada plugin, y por lo tanto, sabremos la versión del mismo que estamos usando en nuestro proyecto.

El elemento **Dependencies** nos muestra todas las librerías que utiliza nuestro proyecto (bajo la etiqueta **<dependencies>** de nuestro pom.xml).

Ejercicios

Para realizar los ejercicios de esta práctica proporcionamos, en el directorio **Plantillas-P01**, un proyecto maven (directorio **P01-IntelliJ**).

Para poder hacer los ejercicios necesitarás:

- **CLONAR** tu repositorio de Bitbucket en algún directorio, por ejemplo, supongamos que lo haces en el directorio \$HOME/practicass. En ese caso, tu directorio de trabajo, en el que debes hacer los ejercicios, será: **\$HOME/practicass/ppss-2021-Gx-apellido1-apellido2**.

IMPORTANTE!!! RECUERDA que a partir de ahora, TODO lo que hagas en prácticas estará en algún subdirectorio de tu directorio de trabajo.

- **COPIA** el directorio **P01-IntelliJ** en tu directorio de trabajo y sitúate en él. A partir de aquí, se pide:

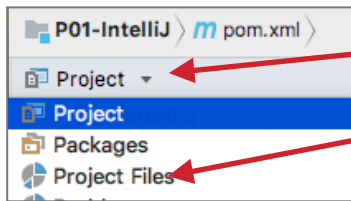
⇒ Ejercicio 1

Activa tu licencia en IntelliJ. Abre el proyecto Maven P01-IntelliJ a partir del directorio que contiene el pom.xml (opción Open). Asegúrate de que la ventana **Maven** está visible. Comprueba que la plataforma SDK, versión 11 está seleccionada, desde **File→Project Structure→Platform Settings→SDKs**, y que el proyecto tiene asignada dicha versión (desde **File→Project Settings→Project→Project SDK**).

A continuación realiza lo siguiente:

Maven
Directory
Layout

- A) Observa la **estructura de directorios del proyecto**. La información del proyecto puede mostrarse desde diferentes "perspectivas". Si quieres ver exactamente las carpetas físicas del disco duro debes mostrar la vista "Project Files", en lugar de "Project", que es la que tendrás por defecto (ver **Figura 1**). Puedes anotar dicha estructura en un fichero .txt para facilitar tu proceso de estudio.



La vista **Project** NO muestra la estructura física de directorios.

Para ello deberemos cambiar a la vista **Project Files**

Figura 1. Un mismo proyecto se puede "visualizar" de formas diferentes.

La estructura de directorios es la misma en CUALQUIER proyecto Maven. Es importante conocerla, ya que el proceso de construcción que realiza Maven asume que determinados artefactos están situados en determinados directorios. Por ejemplo, si el código fuente de las pruebas lo implementásemos en el directorio /src/main/java, no se ejecutarían dichos tests aunque lanzásemos la fase "test" de Maven

Maven
pom.xml

- B) Muestra en el editor la configuración de nuestro proceso de construcción (**fichero pom.xml**). Verás que el fichero xml contiene información sobre: las coordenadas, propiedades, dependencias y sobre la construcción del proyecto (etiqueta <build>)

Nuestro pom define las coordenadas de nuestro proyecto, y además hace referencia a dos artefactos: uno en la sección de dependencias y dos en la sección <build>, cada uno de los cuales tiene sus propias coordenadas. Deberías saber identificarlas. Recuerda que cualquier artefacto (fichero) generado y/o usado por maven se identifica por sus coordenadas. Nuestro proyecto maven se empaquetará de alguna forma y se generará el correspondiente artefacto, por eso tenemos que proporcionar las coordenadas de nuestro proyecto maven.

Fíjate también que los artefactos usados en nuestro pom, se usan en secciones (etiquetas) diferentes, y que cada sección tiene un propósito diferente, y por lo tanto los tipos de ficheros (artefactos) pueden ser también diferentes.

La etiqueta **<properties>** se utiliza para definir y/o asignar/modificar valores a determinadas "variables" usadas en nuestro pom.xml. Podemos usar propiedades ya predefinidas (por ejemplo la propiedad "project.build.sourceEncoding"), o podemos definir cualquier propiedad que nos interese. A partir de Maven 3 es OBLIGATORIO especificar en el pom.xml un valor para la propiedad "project.build.sourceEncoding", por lo que esta línea aparecerá en todos los ficheros pom.xml de nuestros proyectos.

Observa que hemos especificado el valor "test" para la etiqueta **<scope>** en uno de los artefactos. Dicho valor indica que el artefacto en cuestión sólo se necesita durante la compilación de los tests. Si esta etiqueta se omite, su valor por defecto es "compile" y significa que el artefacto es necesario para compilar los fuentes del proyecto.

El pom.xml de nuestra construcción incluye el plugin **maven-surefire-plugin**. Este plugin ya está incluido por defecto cuando usamos el empaquetado jar. Lo que ocurre es que la versión que se incluye por defecto es anterior a la 2.22.0 (que es la versión mínima requerida para poder compilar nuestros tests con junit5). Para ver qué versión viene incluida por defecto debes comentar el plugin en el pom, pulsar sobre el primer icono a la izquierda de la ventana Maven Projects("Reload All Maven Projects"), y consultar la versión del plugin desde dicha ventana. Recuerda que un comentario xml empieza por "<!--" y termina por "-->".

También hemos incluido el plugin **maven-compiler-plugin**. Éste ya está incluido por defecto, pero necesitamos una versión posterior, en concreto usaremos la versión 3.8.0 para poder compilar usando jdk 11, que hemos instalado en la máquina virtual. Averigua qué versión se incluye por defecto de este plugin.

Maven
Source
Code

- C) Con respecto al **código fuente**, la **clase Triángulo** contiene la implementación de un método cuya especificación asociada es la siguiente: Dados tres enteros como entrada, que representan las longitudes de los tres lados de un triángulo, y cuyos valores deben estar comprendidos entre 1 y 200, el método tipo_triángulo devuelve como resultado una cadena de caracteres indicando el tipo de triángulo, en el caso de que los tres lados formen un triángulo válido. El tipo puede ser: "Equilátero", "Isosceles", o "Escaleno". Para que los tres lados proporcionados como entrada puedan formar un triángulo tiene que cumplirse la condición de que la suma de dos de sus lados tiene que

ser siempre mayor que la del tercero. Si esto no se cumple, el método devolverá el mensaje “No es un triángulo”. Si alguno de los tres lados: a, b, ó c, es mayor que 200 o inferior a 1, el método devolverá el mensaje “Valor x fuera del rango permitido”, siendo x el carácter a, b, ó c, en función de que sea el primer, segundo, o tercer valor de entrada el que incumpla la condición, y con independencia de que los tres lados formen o no un triángulo.

Este es uno de los ejemplos más utilizados en la literatura sobre pruebas, quizá porque contiene una lógica clara, pero a la vez compleja. Fue utilizado por primera vez por Gruenberger en 1973, aunque en una versión algo más simple.

Fíjate que esta especificación nos proporciona el conjunto S que hemos visto en la sesión de teoría.

- D) La clase **TrianguloTest** contiene la implementación de cuatro casos de prueba (los cuatro métodos anotados con `@Test`) asociados a la especificación del apartado anterior. Después de estudiar la teoría deberías ser capaz de identificar dichos casos de prueba, y crear la correspondientes tabla . Puedes identificar cada caso de prueba como C1, C2, C3, y C4, y deberías tener claro cuántas columnas necesitas en la tabla y lo que significa cada una de ellas.

Maven
Test
Code

Identificador del Caso de prueba	Dato de entrada 1	...	Dato de entrada n	Resultado esperado
----------------------------------	-------------------	-----	-------------------	--------------------

Observa la implementación de cada test y verás que todos ellos siguen la misma lógica de programa. Puedes **anotar el algoritmo** que refleja dicha lógica. Recuérdalo porque lo utilizaremos también en sesiones posteriores.

➡ Ejercicio 2

Vamos a “construir” el programa. En este caso sólo vamos a compilarlo. Para ello haremos doble click sobre la fase “**compile**” desde la ventana “Maven Projects”. Esta acción en el IDE es equivalente a ejecutar desde línea de comando la orden: `mvn compile`. Puedes comprobar que IntelliJ ejecuta la versión de maven que hemos instalado en `/usr/local` de nuestra máquina virtual. Esto lo puedes hacer desde la ventana “Maven”, pulsando sobre el icono que tiene un dibujo de una llave inglesa. O desde las preferencias del IDE (*Build, Execution, Deployment*→*Build Tools*→*Maven*). **Importante:** si cuando consultes la ruta de Maven (“Maven home path”) aparece “Bundled Maven”, cámbialo por la ruta `/usr/local/apache-maven-3.6.3`. Cuando ejecutes “`mvn compile`”, el resultado se muestra automáticamente en una ventana en la parte inferior del IDE en donde verás los mensajes que genera Maven durante el proceso de construcción.

Maven
Build
(compile)

- A) Observa que ha aparecido un directorio nuevo en nuestro proyecto maven (en el panel de la izquierda): el directorio target. Fíjate en la nueva estructura de directorios creada, y qué artefactos contienen. Esta nueva estructura, así como la ubicación de los artefactos también es común para CUALQUIER proyecto maven, por lo que deberás conocerla. Ahora vamos a ejecutar la fase “**clean**”. Observa lo que ocurre y fíjate en las goals que se ejecutan. Ahora vuelve a compilar el proyecto. Fíjate en la secuencia de acciones que se muestran en la ventana inferior y en que NO se han ejecutado los tests

Maven
Build
(clean)

- B) Para **ejecutar los tests** vamos a hacer doble click sobre la fase “**test**”. Fíjate en la salida por pantalla, concretamente (y de momento) nos interesa la siguiente información (pulsas primero sobre el elemento P01-IntelliJ de la ventana Run para ver la pantalla de *logs* de Maven):

Maven
Build
(test)

“**Tests run**” indica el número total de tests ejecutados. “**Failures**” indica el número de tests cuyo resultado esperado NO coincide con el real. Observa que junit proporciona un tercer tipo de resultado: “**Error**”, del que hablaremos en sesiones posteriores. Verás que uno de los tests falla, es decir representa un fallo de ejecución (*failure*). Esto significa, como ya hemos indicado, que el valor del resultado esperado y el real NO coinciden. De hecho vemos también que por pantalla se muestra la razón del fallo de ejecución del test. Observa también algo muy importante, el resultado de la construcción es: **BUILD FAILURE**, es decir, el proceso de construcción (`mvn test`) no se ha completado con éxito puesto que se han detectado problemas en la ejecución de alguna de las goals del proceso, concretamente en la fase de pruebas.

Alternativamente, podemos ver los resultados de ejecución de los casos de prueba de forma gráfica seleccionando, desde la **ventana Maven**, el icono con una **M de color rojo** (a la derecha de la llave inglesa). Esto mostrará la ventana “Maven Test Results”, en donde vemos más claramente qué test es el que ha fallado, y cuál es el resultado esperado y el real obtenido. Nota: inicialmente sólo se

muestran los tests con failures. Podemos mostrar también los tests que han pasado con éxito seleccionando el icono “√” de la ventana Maven Test Results. Observa que cuando ejecutamos la fase “test” de maven se ejecutan TODOS los tests (es decir, también el de MatriculaTest).

debugging

- C) Para poder concluir nuestro proceso de construcción con éxito: BUILD SUCCESS, necesitamos eliminar el problema/s que provoca el fallo de ejecución. En este caso se trata de averiguar por qué el informe del resultado de la ejecución del caso de prueba correspondiente es un fallo. Hemos cometido un error en la implementación del método que estamos probando, que hace que el resultado esperado (resultado que debería dar si estuviese bien implementado) no es el real. Identifica la causa y modifica el código para que el resultado real sea el correcto (recuerda que este proceso se llama DEPURACIÓN, o *debugging*). A continuación vuelve a ejecutar la fase test. **Tendrás que volver a pulsar sobre el icono con una M para ver actualizados los resultados de forma gráfica**. Repite el proceso hasta que los cuatro tests estén en “verde”, y el proceso de construcción termine con: BUILD SUCCESS.

Test Case Design

- D) Observa qué tienen en común el test C1 y un posible test adicional C5 con datos de entrada: a=7,b=7,c=7, y piensa en la conveniencia o no de incluir C5 al conjunto de tests. De la misma forma razona si son necesarios los tests C2 y C3. Añade dos posibles casos de prueba adicionales que “aporten valor” al conjunto de casos de prueba (no sean innecesarios).

⇒ Ejercicio 3

La clase **Matricula** contiene el método **calculaTasaMatricula()** que devuelve el valor de las tasas de matriculación de un alumno en función de la edad, de si es familia numerosa, y si es o no repetidor, de acuerdo con la siguiente tabla (asumiendo que se aplican sobre un valor inicial de tasa=500 euros):

	Edad < 25	Edad < 25	Edad 25..50	Edad 51..64	Edad ≥ 65
Edad	SI	SI	SI	SI	SI
Familia Numerosa	NO	SI	SI		
Repetidor	SI				
Valor tasa-total	tasa + 1500	tasa/2	tasa/2	tasa -100	tasa/2

Test Case Design

- A) En este caso, hemos proporcionado la implementación de un único test, en la clase MatriculaTest. Rellena una tabla de casos de prueba con el test que hay implementado y piensa 5 nuevos tests que no sean “redundantes” y añádelos a la tabla. En tu opinión, ¿6 tests son suficientes o deberíamos añadir alguno más?

Test Case Code/Run

- B) Implementa los casos de prueba que has añadido a la tabla y ejecuta todos los tests. Recuerda que si encuentras algún error debes depurarlo.

Observa que primero (apartado A), tenemos que obtener (decidir) los casos de prueba que usaremos para comprobar que el programa funciona correctamente. Para ello tenemos que “elegir” datos de entrada concretos, y asociar un resultado esperado, de acuerdo con el comportamiento correspondiente de la especificación.

A continuación (apartado B) tenemos que implementar los tests, y finalmente ejecutarlos. Cuando obtengamos el informe de JUnit podremos ver si hemos detectado defectos en nuestro programa o no. Es muy importante que te quede claro que un informe JUnit sin errores no significa que el programa esté libre de ellos. Es decir, nuestras pruebas sólo pueden demostrar la presencia de errores (no la ausencia de los mismos).

➡ Ejercicio 4

Hasta ahora hemos lanzado la ejecución de las fases del ciclo de vida de maven “clean”, “compile” y “test”. Vamos a ejercitar otras dos fases importantes: la fase “package”, y la fase “install”.

**Maven
Build
(package)**

A) Ejecuta la **fase “package”** y observa los cambios en la ventana del proyecto. Fíjate en qué acciones se han llevado a cabo para construir el proyecto y qué artefactos nuevos se han generado. Es importante que tengas claro qué artefactos se generan en cada fase, para poder utilizar de forma adecuada maven. Modifica uno de los tests, de forma que dé un resultado fallido, ejecuta la fase “clean”, y a continuación la fase “package” de nuevo, y observa lo que ocurre. De igual forma, ahora, en lugar de introducir un error en los tests, edita el fichero Matricula.java, quita un punto y coma para provocar un error de compilación y vuelve a ejecutar las fases “clean” y “package”. Tienes que tener claro el resultado obtenido y por qué se obtiene dicho resultado.

**Maven
Build
(install)**

B) Vuelve a reparar todos los errores introducidos y ejecuta la **fase “install”**. En este caso el resultado es menos “obvio” ya que en esta fase se “instala” (copia) el artefacto generado en la fase anterior en el repositorio local. El repositorio local de maven se encuentra en \$HOME/.m2/repository. En el repositorio local se almacenan todos los artefactos que maven ha utilizado para construir el proyecto, más todos aquellos artefactos que hayamos “instalado”, por ejemplo, utilizando la fase install. Deberías saber la ruta exacta de cada artefacto a partir de sus coordenadas. Si borras el directorio .m2 no importa, maven lo volverá a crear automáticamente durante la próxima ejecución del comando mvn. La primera vez que ejecutemos maven, si el repositorio local está vacío, maven se descarga todos aquellos ficheros que necesita de sus repositorios remotos. Esto significa que, a medida que vayas ejecutando maven y necesitando los artefactos (ficheros jar, war, ear, pom,...) para construir el proyecto, tu repositorio local irá creciendo. Si maven encuentra en el repositorio local el artefacto correspondiente, no será necesario proceder a su descarga, por lo que se reducirá el tiempo de construcción del proyecto.

Guardamos nuestro trabajo en Bitbucket

Recuerda que debes guardar TODO tu trabajo de prácticas en Bitbucket. Deberías “subir” a Bitbucket los ejercicios según los vas realizando (no esperes a tenerlos todos, podrías perder tu trabajo si tienes algún problema con la máquina virtual).

Para ello simplemente debes usar los comandos git que ya hemos visto, desde un terminal y **desde tu directorio de trabajo** (ppss-2021-Gx-apellido1-apellido2):

**Git/
Bitbucket**

- git add .
- git commit -m“Ejercicio P01-X terminado”
- git push

Resumen



¿Qué **conceptos** y **cuestiones** me deben quedar CLAROS después de hacer la práctica?

TEORIA

PRACTICA

GIT

- Herramienta de gestión de versiones. Nos permite trabajar con un repositorio local que podemos sincronizar con un repositorio remoto.

MAVEN

- Herramienta automática de construcción de proyectos java.. El “build script” se especifica de forma declarativa en el fichero pom.xml, en el que encontramos varias partes bien diferenciadas. Los artefactos maven generados se identifican mediante sus coordenadas.
- Los proyectos maven requieren una estructura de directorios fija. El código de los tests está físicamente separado del código fuente de la aplicación.
- Maven usa diferentes ciclos de vida. Cada ciclo de vida está formado por una secuencia ordenada de fases, cada fase puede tener asociadas unas goals. El resultado del proceso de construcción maven puede ser “Build fauilure” o “Build success”.

TESTS

- Un test se basa en un caso de prueba, el cual tiene que ver con el comportamiento especificado del elemento a probar.
- Una vez definido el caso de prueba para un test, éste puede implementarse como un método java anotado con @Test (anotación JUnit). Los tests se compilan y se ejecutan en diferentes momentos durante el proceso de construcción de un proyecto.
- El algoritmo de un test es siempre el mismo. Un test comprueba, dadas unas determinadas entradas sobre el elemento a probar, si su ejecución genera un resultado idéntico al esperado (es decir, se trata de comprobar si el comportamiento especificado en S coincide con el comportamiento implementado P)
- Dependiendo de cómo hayamos diseñado los casos de prueba, detectaremos más o menos errores (discrepancias entre el resultado esperado y el resultado real).